An AOP Layer to Abstract Programming with Distributed Components

Fabrice Legond-Aubry, Gérard Florin, Lionel Seinturier

An AOP Layer to Abstract Programming with Distributed Components

F. Legond-Aubry [1], G. Florin [2]

Laboratoire CEDRIC
Conservatoire National des Arts et Métiers,
292 rue Saint Martin,
75005 Paris,
France

L. Seinturier [3]

Laboratoire d'Informatique de Paris 6
Université Pierre et Marie Curie
4 place Jussieu,
75252 Paris cedex 5,
France

Abstract

Component models such as EJB or CCM are increasingly used to create complex and distributed systems and applications. Unless the details of the API and mechanisms used for programming with these models differ, the general features provided by the models share many characteristics. In this paper, we propose to capture these features in a common layer that can be used to develop model independant component programs. The layer is then mapped either onto EJB or CCM. This layer is defined with the AspectJ aspect-oriented programming language. We report on two samples applications that were developped with this layer: an application-level load balancing service and a contract enforcement service.

Key words: AOP, Contract, Assembly, Component, Constraint, Interaction, Model

[1] Email: `fabrice.legond-aubry@lip6.fr`
[2] Email: `florin@cnam.fr`
[3] Email: `lionel.seinturier@lip6.fr`

1 Introduction

Component-oriented middleware platforms such as EJB [18] or CCM [15] share many characteristics. They provide a model for distributed component programming. They are based on a Remote Procedure Call (RPC) mechanism (in both cases based on the IIOP protocol). Their server-side architecture is based on a container that host components and provide system-level services (for persistence and security for instance). Finally, their API altough different, share many common features. Hence, programming with either of these environments is very similar.

The general idea underlying this paper is to provide a common layer to encompass approaches for programming with distributed components such as EJB or CCM. This layer will allow to develop applications that will run either on the two platforms. Also, even if in a first approach we only take into account these two models, we don't want to restrict our investigations and we want to design a solution that could be adapted to support other existing (e.g. .Net, Fractal [3]) or future component models.

Many solutions could be set up to meet this goal:

(i) we could work at the design level and set up a general meta-model for components models and, in the MDA fashion, define projections towards component technologies;

(ii) we could work at the program level, and provide some common programming abstractions that would fit both models;

(iii) we could work at the protocol level and set up some gateways between the two technologies.

Solution (i) is out of the focus of this paper. This was the subject of a previous project whose results were presented in [5]. The meta-model was defined for the Objecteering UML CASE tool from the Softeam company and the projection was defined with transformation rules written in the J programming language of the Objecteering UML CASE tool. Solution (iii) is too low level. It requires to run both application servers as soon as one needs a given personality of a component developed for the other technology. The focus of this paper is on solution (ii). Contrary to (i) where the mapping from model to code was developed on a ad-hoc manner, we want to provide some common programming abstractions that allow us to develop component-oriented distributed applications either with EJB or CCM. Thus this work complements (i) and will ease the development of the projection code.

To provide these common programming abstraction, several choices can be made: we could provide an API with a class library, design patterns to adapt component to a given personality or we could use code transformation library such as the ones provided by BCEL for the Java bytecode. Our final choice has been to investigate an aspect-oriented approach. The idea is to let the component code free of any intrusion by specificities imposed by the

component platform such as EJB and CCM, and to define and implement aspects that enhance this code to obtain either EJB code or CCM code.

The main aim of this article is to define a generic abstract aspect-oriented layer for distributed components platforms. To do so, we locate the core cross-platform events and behaviors and capture them in a set of AOP pointcuts. Thanks to them, we implement some application-level crosscutting services like fault-tolerance, load-balancing and contracts without the need of knowing the internal mechanisms of a particular platform.

This paper begins with a short state of the art on component models and AOP (section 2). Then we expose in section 3 common events and behaviors in some component frameworks (EJB, CCM) From this, we define the AspectJ aspects and pointcuts that form the core of our layer (section 4). As an example of the viability of the solution, we report in section 5 on some experiments that consist in implementing load balancing and component contracts in the EJB JOnAS platform. Finally, section 6 concludes this paper and provides some perspectives for this work.

2 AOP and Components paradigms

Components platforms were introduced to alleviate some of the object oriented model defaults. In particular, the problem of the instances thigh binding, the distribution and the concerns mixin. They introduce some pre-implemented services like persistence and transaction to ease the developers task in creating distributed applications. The granularity and the offered functionalities of the components are more important due to the embedded middleware code: they become small servers that render functional services.

The "Entreprise Java Beans" (EJB) was proposed by Sun Microsystems in 1999. It defines a conceptual canvas to make distributed application by deploying small server entities. EJB applications are basically composed of:

- EJB servers that provide distant invocable services.
- EJB containers that encapsulate the EJB servers, manage them, and provide some add-hoc non-functional services.
- EJB clients applications that use the EJB servers and containers.

In the EJB model, multiple servers types can be instantiated:

- Message driven component that implement message reactive code only
- Session component that can be stateless or statefull that have no persistence.
- Entity component that have embedded persistence.

The Object Management Group use the same concepts as EJB to built Corba Component Model (CCM). The CCM model splits the application development in different processes: the programmer that focus on the component functionalities, the assembler that make components assemblies and the

packager who deploy non-functional services. CCM also haves the servers, containers and clients entities.

CCM defines six containers kinds, accordingly to the components needs: Session containers (stateless or state full) for components that have no persistence, Entity Containers for persistent components that represent persistent elements (eg. data in a database), and so on. CCM also allows multiple component implementations for one component abstract view.

From the architectural point of view, it is an advance but it leaves little room for non functional services extensions. It is hard to add platform wide new service (like replication) and coding them for each application should not be considered as a viable solution. So, the separation of between concerns are insufficient.

A first solution to implement new middleware services is a pure component oriented solution. You can develop an extensible platform where you can plugin/plugout services like in JavaPod [2] or OpenCorba [9]. But Interoperability with current and old platforms is lost. Also the utilisation of such extensions in an industrial context is limited because major firms prefer rock solid solutions on new non proof-of-concept solutions.

A second solution is to extend existing models and purely map it to standard models via predefined transformations (design patterns) like in MDA. MDA aims to provide platform independent model and means to project it on existant platform. It also provide design level transformations which should be automatically mapped into code. Thus, each new extension is automatically transformed in an predesigned components assembly. Improving a component model this way can be efficient in term of time because it requires only few additional knowledge to deploy this solution but it render up the model of an not yet mastered technology harder to use. However it has the great advantage to maintain full backward compatibility [12].

Another alternative is to implement new services by modifying components. E. Truyen et al. [20] uses wrappers but AOP offer more complex and efficient high level ways of expressing dynamic events during the development cycle.

AOP modularizes OOP even more by separating the extra functionalities from an applications primary functionalities. AOP does not replace OOP or components but instead makes it more efficient by implementing crosscutting concerns. One of AOPs main goals is to define appropriate situations during the code execution where a specific external code (called *aspect*) will apply. *Pointcuts* designate to the places in the application code, the *joinpoints*, where the desired functionality needs to be inserted. The *weaving* process is the act of inserting the aspects in the application code.

Thus AOP enable easy multi-platform mono-language integration of extensible services at application level. Currently, Java is the most widely used language for components. So, using a Java AOP framework can help to implement a lighter (in term of code to write), more efficient (in term of speed),

inter-operable solution than a component based one. AOP Alliance [1] will soon offer a standardized way of expressing AOP concepts and its potential linking with J2EE.

AspectJ [8] is a compilation time extension of Java that enables user to inject aspects at the compilation time. It has less capacities than dynamic ones but is easier, offer a good support and is more suited for developments. It works on the source code, have less side effects than dynamics ones and can be properly embedded in component code. This scheme best fit the applications conception point of view because it works on the source code and not on the compiled code. It enables clean code manipulations and clean compiled code generation.

Dynamic AOP platforms enable aspects weaving at runtime. This is done by swapping the Java Virtual Machine standard class loader by a new one that patches, thanks to bytecode translators tools, the application classes. They add the specific necessary code used for aspects weaving. However, components platforms use multiple Java classloaders to strengthen isolation between components. So, the dynamic AOP classloaders and bytecode translators interfere with the components platform normal behavior. It should be corrected in a near future. AspectWerkz [21] or RtJAC [4] are a step toward this because they act at low level in the JVM but they use undocumented tricks of the Sun VM. The major drawback of these dynamic platforms is that they are not generic enough. Whereas source oriented AOP platforms can be used for multiple components middleware, multiple JVM, dynamic AOP platforms as a scope limitation that lies in its heavy modification of the executed code.

To add credits to this proposal, JBOSS AOP [19] hold an aspect interface called JBOSS AOP. In JBoss, simple Java objects can have features such as transactions and security that are usually reserved for EJB components. These features are provided as a collection of pre-coded aspects, and can be applied to any application object via dynamic weaving, without requiring the application itself to be recompiled. In other words, the JBoss AOP framework allows developers to write Java components and apply ad-hoc services later on in the development cycle without changing a line of of the application Java code. However it is tightly integrated to the JBoss application server and, then again, limits its scope.

3 Components events and behavior

The general idea underlying this paper is to provide an aspect layer that encompasses common mechanisms for programming with distributed components such as EJB or CCM. This layer will allow to develop applications that will run either on the two platforms.

Component oriented middleware platforms such as EJB or CCM share many similarities. They all have nearly the same structure. Small differences are introduced by the components type and platforms but the analysis still

holds. We have identified four major basic steps when programming with such platforms:

(i) Middleware binding phase
(ii) Component binding phase
(iii) Invocation phase
(iv) Interaction termination phase

Figure 1 illustrates these steps with some sequence diagrams.

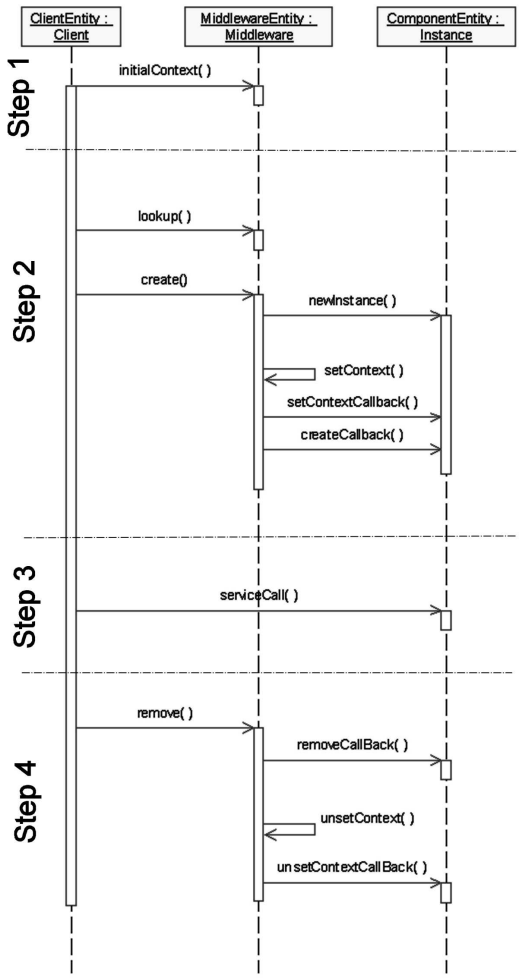

Fig. 1. Component generic behaviors

The *initial service discovery* is the first mandatory step for any component service invocation. In this step, the client gets a local proxy for the middleware, so that it can have access to the basic services like the naming lookup service. The second step (*component binding*) is to get a proxy object for the

component. It is also a mandatory preliminary step to make the real invocation. The next step is the real *invocation* phase. Eventually, the client can close its connection to the component instance in the *interaction termination* step by using the component or middleware proxies. Concurrently to the last three steps, a client can call some component platform services. Steps 2 and 4 are also part of them. Apart from these steps, the component platform can act on the components thanks to the container. These events are not included in the interaction and will be discussed in the last part of this section (3.5).

3.1 Step 1 : Initial Services Discovery

In the EJB and CCM platform, the connection to the middleware is very simple. We just bootstrap it by making a static call on an existing class or by creating a new object that we will be the middleware proxy to interact with (listing 1). One of the most obvious service can be assimilated as a factory of component factories via a global lookup service connection.

Listing 1: EJB/CCM middleware binding

```
//EJB platform
Context ctx = new InitialContext(env);

//CCM platform
chf = CORBA.ORB.resolve_initial_references ("ComponentHomeFinder");
```

These events are interesting to capture because they allow to implement middleware bridges. It is also a good start for implementing security services.

3.2 Step 2 : Component Binding

The second step is to get a reference on a component factory (i.e. the Home interface of EJB/CCM). This is mostly done by a lookup operation (listing 2). An optional further instruction is the *narrowing* of the component reference. This operation casts the component. From this point, we have a stub "object" on which we can invoke services.

Listing 2: EJB/CCM component binding

```
//EJB platform
Object obj=ctx.lookup("HelloWorld");
HelloWorldHome home=(HelloWorldHome)javax.rmi.PortableRemoteObject.narrow
                  (obj, HelloWorldHome.class);
HelloWorld helloWorld=home.create();

//CCM platform
HelloWorldHome=chf.find_home_by_type(HelloWorldHome.id());
HelloWorldManager=HelloWorldHome.create();
helloWorld=HelloWorldManager.provide_HelloWorld ();
```

These events are interesting to capture because they enable load balancing, or potentially dynamic stub swapping. Dynamic stub swapping is useful for changing the communciation layer between a component instance and its client.

3.3 Step 3 : Invocation

The interaction is encapsulated in an interaction context. The context is created and handled by the middleware, and not directly by the programmers that can nevertheless access it. The invocation of the service is seen as a local call on a local object which is a stub for the remote component. However, distribution do not imply full-transparency. For instance, distribution related exceptions are associated to interactions and must be caught. Figure 2 sums up the main distribution related exceptions that can be found in EJB and CCM.

Action	EJB	CCM
Middleware binding error	NamingContext	InvalidName
Lookup error	FinderException, ObjectNotFoundException	FinderFailure, UnknowKeyValue
Creation error	CreateException, DuplicateKeyException	CreationFailed, DuplicateKeyValue
Remove error	RemoveException	RemoveFailure
Invocation error	EJBException, RemoteException	CCMException

Fig. 2. Exception for components framework

These interaction events are interesting to capture because they enable faut-tolerance or security on the component. They also enable to implement behavioral adaptation for the client and servers. Another possibility is the dynamic delegation of services. In our second example, intercepting component methods call let us check the validity of the call at the client and the server side.

3.4 Step 4: Invocation Termination

To close the invocation phase, the client calls a remove or a passivate method on the component or the middleware to end the transaction. Sometimes, the clients skip this step because it is not mandatory: they let the middleware close the connection at the end of a timeout timer.

3.5 Containers related events

EJB and CCM provide non-functional services such as transactions, security and data persistence With these services the behavior of components can be transparently extended, provided that the developer sets the needed values in some XML parameter description files. For instane, with the persistence service, components can be loaded and stored into a database. Some system event are made available to the components through callback methods. Figure 3 sums up the callback methods that can be found in the EJB and CCM platforms.

Callback	EJB	CCM
creation	ejbCreate	implemented on home
remove	ejbRemove	ccm_remove
context setting	setXXXContext	set_XXX_context
context freeing	unsetXXXContext	unset_XXX_context
activation	ejbActivate	ccm_activate
passivation	ejbPAssivate	ccm_passivate
loading	ejbLoad	ccm_load

Fig. 3. Middleware callbacks

4 Implementing the aspect layer for EJB components

For our example, we pick up the AspectJ framework. AspectJ is currently our choice because of its genericity and its integration in the Eclipse IDE environment, its maturity and the efficiency of its generated code. As for the EJB platform, we choose the open source JOnAS [13] from the ObjectWeb consortium. The ObjectWeb consortium also provides the OpenCCM [14] implementation of the OMG CCM specification on which we plan to extend our study. Note that the solution could also be applied to other EJB platforms with only very minor adaptations.

The following sub-sections define the AspectJ pointcuts that correspond to the 4 four steps defined in the previous section. The definition of these pointcuts is given here for the EJB platform. Due to some lack of space, we skip the definition of these same pointcuts for the CCM platform. These pointcuts can then be reused to implement component platform independant applications. Section 5 illustrates these issues

4.1 Step 1 : Initial Services Discovery pointcuts

In EJB, we just trap the first client invocation to the naming service (Figure 4).

Pointcut name	Side	Pointcut expression
serverside_middlewareBinding()	server	ϕ
clientside_middlewareBinding()	client	call (javax.naming.InitialContext.new (..));

Fig. 4. The Middleware binding pointcuts

4.2 Step 2 : Component Binding pointcuts

For the component binding step, we trap three specific actions (Figure 5): the client lookup (*lookup* or *getEJBObject* methods), the component narrowing (the *narrow* method), the connection to a component instance (the *create*, *findByPrimaryKey* methods). There is no server side lookup because the component is loaded by the middleware. Deployment is done thanks to the parsing of the XML component configuration file. We could trap the *newInstance* or the bean static initialization to catch the server side creation. When a component make a lookup, it takes the client role so there is no server side lookup

pointcut. The *serverside_create* and *serverside_setContext* pointcuts traps the callbacks execution. They are mandatory in any EJB beans.

Pointcut name	Side	Pointcut expression
clientside_lookup()	client	call (* javax.naming.Context.lookup (String));
serverside_lookup()	server	ϕ
clientside_narrowing()	client	call (* javax.rmi.PortableRemoteObject.narrow (..));
serverside_narrowing()	server	ϕ
clientside_create()	client	!withinEJBMiddleware() && call(* javax.ejb.EJBHome+.create (..));
serverside_create()	server	execution(* javax.ejb.*Bean+.ejbCreate(..));
clientside_setContext()	client	ϕ
serverside_setContext()	server	call(void javax.ejb.*Bean+.set*Context (..));

Fig. 5. Some of the components binding pointcuts

4.3 Step 3 : Invocation pointcuts

In the invocation phase (figure 6), we choose to exclude all invocations to the middleware callbacks methods. We exclude them from this pointcut because they are not really part of the application interaction. For attributes access, we trap methods beginning with "*set*" and "*get*" and carefully avoiding *setXXXXContext* methods.

Pointcut name	Side	Pointcut expression
clientside_componentCall()	client	(call(* javax.ejb.EJBObject+.*(..)) || call(* javax.ejb.EJBHome+.*(..))) && !(withinEJBMiddleware());
serverside_componentCall()	server	execution(public * javax.ejb.*Bean+.*(..)) && !execution(public * javax.ejb.*Bean+.ejb*(..));
serverside_componentAttributeSet()	server	execution (public void javax.ejb.*Bean+.set*(*)); && !execution(void javax.ejb.*Bean+.set*Context (..));
clientside_componentAttributeSet()	client	call(public void javax.ejb.*Bean+.set*(*)) && !call(void javax.ejb.*Bean+.set*Context (..));
serverside_componentAttributeGet()	server	execution (public void javax.ejb.*Bean+.get*(*));
clientside_componentAttributeGet()	client	call(public void javax.ejb.*Bean+.get*(*));

Fig. 6. Invocation pointcuts

4.4 Step 4: The invocation termination pointcuts

The end of the interaction, can be done by unsetting the context of interaction and by optionally make a call to the container *remove* and *unsetXXXContext* methods (figure 7).

Pointcut name	Side	Pointcut expression
clientside_remove()	client	call(* javax.ejb.EJBHome+.remove (..));
serverside_remove()	server	execution(* javax.ejb.*Bean+.ejbRemove(..));
serverside_unsetContext()	server	call(void javax.ejb.*Bean+.unset*Context(..));

Fig. 7. Invocation termination pointcuts

4.5 The Containers and the middleware pointcuts

They are several different middleware-component interactions depending the component kinds. On a EJB platform, we catch these interaction by trapping all calls to "*ejbXXX*" methods. Some of them can raise false and cause behavioral havoc when then programmer name one of its functional method "ejbXXX". Subtler pointcut could be defined to avoid this.

Pointcut name	Side	Pointcut expression
serverside_middlewareCall()	server	execution(* javax.ejb.*Bean+.ejb* (..));
clientside_middlewareCall()	client	ϕ

Fig. 8. Some Middleware-Component callbacks pointcuts

5 Sample applications : Load balancing and dynamic contracts checking

5.1 An EJB specific load balancing service

One first example is to set a load balancing service. To do this, we create an aspect that extends our abstract layer aspect which will catch the middleware binding event. As we don't want to overload the client, we just grab the opened connection of the client when it connect to the middleware (listing 3)

Listing 3: Middleware binding interception

```
after() returning (InitialContext ic): clientside_middlewareBinding() {
    // Get the connection object reference
    initialContext=ic;
}
```

After, we the client need to bind a component for a service, we intercept the request, lookup for replicates for potential future interactions (listing 4).

Listing 4: Component binding interception

```
before(String EjbJndiName) : clientside_lookup() && args(EjbJndiName) {
    // Intercept all components lookups
    if (EjbJndiName==null)
        return;
    if (EjbJndiName.endsWith("OpHome")) {
        getOpReplicates();
}}
```

We do the load balancing at the component connection (*clientside_creation* pointcut). The client call the create method on the home reference to get a stub for interaction. The component connection request is forwarded or redirected to a component replicate by making a reflexive call on the stub (listing 5).

Listing 5: Instance component creation interception

```
Object around(Object p, Object homeObject) throws RemoteException:
        clientside_creation() && args (p) && target (homeObject)
{
   // if we need an opHome stub, get a replicate else forward the call unmodified
   EJBHome home=null;
```

```
    if (homeObject instanceof OpHome) {
        home = (OpHome)getOneOpHomeReplicate();
    }
    else home = (EJBHome) homeObject;
    // invoke method
    try {
        Method m=home.getClass().getMethod("create",new Class[] {String.class});
        return m.invoke(home, new Object[]{p});
    }
    catch (Exception e)
    { throw new RemoteException(); }
}
```

With this simple code, we have implemented a lookup cache and a load balancing for the Op component. And this is done at the client side so no useless invocation was made. Server side replication could have been done the same way from the server side by capturing the setContext event and redirect invocations accordingly to the caller context but it would have cost a supplemental indirection distant call instead of a local call.

5.2 Components and generic contracts

Contracts were introduced in the object paradigm by Helm [6]. Its goal was to compensate the lack of methods to express the relations between objects. They were used to specify behavioral compositions. Using contracts provides an orthogonal dimension to the one provided by the class structure. To improve these techniques and allow re-use, contracts were extended and adapted to components by Meyer [11] and Jezequel [7]. Generally, contracts exist for server side only or for both the client and the server at the same time. Moreover, contracts for components are limited to specific purpose platform.

Tools like Jcontract [16], Icontract [17] or Eiffel [10] implement "Design By Contract" assertions but for an Object Oriented Model not for components. Moreover they are used for test purpose and not at all for model checking.

Contracts are expressed under the form of automats and properties in a XML file. Separation is a first step, some properties can be check at the design time with some verification tools. But some cannot due to the complexity of the components frameworks and the amount of code involved. So as a beginning part, we choose to make the verification at runtime. More information about the structure of the contracts we use can be found on the Accord project web site[4] . Each client and server should have its own contract. Here a simple version of the server XML contract file (listing 6) that represents the automaton of figure 9 :

Listing 6: XML Contract for the server side

```
<?xml version="1.0" ?>
<!DOCTYPE stateMachine SYSTEM "./statemachine.dtd">
<stateMachine>
    <!-- automat node lists -->
    <beginNode nodeRef="nid_1"/>
    <endNode nodeRef="nid_3"/>
    <node nodeId="nid_1" name="start"/>
    <node nodeId="nid_2" name="accountOpened"/>
    <node nodeId="nid_3" name="end" />
    <!-- transitions lists -->
```

[4] http://www.infres.enst.fr/projets/accord/

```
        <arc source="nid_1" destin="nid_1"><action>setContext</action></arc>
        <arc source="nid_1" destin="nid_2"><action>openAccount</action></arc>
        <arc source="nid_2" destin="nid_2">
            <action>withdraw</action>
            <pre><condition conditionId="cid_10" type="basic">p1>0</condition></pre>
            <post><condition conditionRef="cid_10"/></post>
        </arc>
        <arc source="nid_2" destin="nid_3"><action>closeAccount</action></arc>
    </stateMachine>
```

In order to implement the solution, we have created a component that check contracts from the data collected by the aspect. It will be instantiated by the aspect (Figure 9) when the component is loaded in the EJB middleware (*serverside_staticinit* pointcut) and when a client binds itself to the middleware (*clientside_middlewareBinding*) in the first pointcut of the listing 7. We a connection is made between a client and a server, their respective automats are initialized thanks to the second pointcut. After, until the end of the interaction, we check that the client and server automats fit their own evolution by intercepting all application calls (third and fourth pointcuts). Note that this is only the major code outlines the rest was omitted due to the lack of place.

Listing 7: Aspectj Aspect summarize for contracts checking

```
// Instantiate connection to the component that will check the execution evolution
// Instantiation are done when component are loaded or when a client bind itself
// to the middleware
before() : ( serverside_staticinit() && !within(contractDesign.*) )
            || clientside_middlewareBinding()
    { contractCheckerComponentConnection(); }
// Re-Initialize the automat before a new interaction begin
before() : clientside_create() || serverside_setContext()
    { initializeAutomatForInteraction(); }
// Check contracts before and after service execution
before(): clientsside_componentCall()
    { checkClientSideContract(); }
after(): serverside_componentCall()
    { checkServerSideContract(); }
```

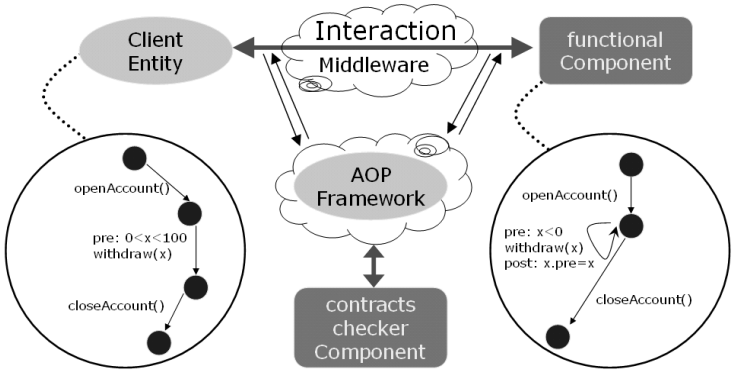

Fig. 9. Client Server component automats

6 Conclusion and future work

We have identified some of the most important component middlewares events. We have defined abstract pointcuts for a specific platform. But the same methodology can be applied for others platforms. We have implemented a

basic aspect that collect information about the caller and callee components and a component contract that check the application calls validity.

In short, we have separated the communication layer from the component structure. So we could easily make generic cross platform communication and services. Other extension are lower level interventions for more specific use of the complex AOP interceptors.

We will to refine the poincuts definitions to fit more precisely to the captured events and reduce potential "wrong catch" side effects. A next step is to complete abstract layer for OpenCCM model and make inter-middleware non-fonctional services. To do so, we need to care of the parameters and make more exemples to evaluate the impact of the component type and the platforms specificities.

Another promising way, is to use AOP to make high level model transformations of distributed components applications thanks to our abstract layer.

References

[1] AOP Alliance. AOP Alliance (Java/J2EE AOP standards). http://aopalliance.sourceforge.net/, 2002.

[2] E. Bruneton and M. Riveill. Experiments with JavaPod, a platform designed for the adaptation of non-functional properties. In *Procedings of Reflection'01*, volume 2192 of *Lecture Notes in Computer Science*, pages 52–72. Springer, September 2001.

[3] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quema, and Jean-Bernard Stefani. An open component model and its support in java. In *Proceedings of the 7th International Symposium on Component-Based Software Engineering (CBSE-7)*, volume 3054 of *Lecture Notes in Computer Science*, pages 7–22. Springer, May 2004.

[4] M. Espak. Improving Efficiency by Weaving at Run-time. In *Generative Programming and Component Engineering Student Workshop*, 2003.

[5] A. Georgin, F. Legond-Aubry, S. Matougui, N. Moteau, A. Muller, A. Tauveron, J.-P. Thibault, and B. Traverson. Description des assemblages et des contrats pour la conception par composants - le projet accord. In *Journes Composants 2004 (JC'04)*, March 2004.

[6] R. Helm, I. M. Holland, and D. Gangopadhyay. Contracts: Specifying Behavioral Compositions in Object-Oriented Systems. In *Proc. of the OOPSLA/ECOOP-90: Conference on Object-Oriented Programming: Systems*, pages 169–180, Languages, and Applications / European Conference on Object-Oriented Programming, Ottawa, Canada, 1990.

[7] J.-M. Jézéquel, M. Train, C. Mingins. *Design Patterns and Contracts*. Addison-Wesley, October 1999.

[8] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An overview of AspectJ. In *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP'01)*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–353. Springer, June 2001.

[9] T. Ledoux. OpenCorba: A reflective open broker. In *Procedings of Reflection'99*, volume 1964 of *Lecture Notes in Computer Science*, pages 197–214. Springer, July 1999.

[10] B. Meyer. *Eiffel : The language*. Prentice-Hall, 1991.

[11] B. Meyer. Applying Design by Contract. *Computer*, October 1992.

[12] M. Blay-Fornarino O. Nano. Using MDA to integrate services in component platforms. In *8th International Workshop on Component-Oriented Programming WCOP 2003*, 2003.

[13] Objectweb Consortium. *The JOnAS project*. `http://jonas.objectweb.org`.

[14] Objectweb Consortium. *The OpenCCM project*. `http://openccm.objectweb.org`.

[15] OMG. *CORBA Component Model*, February 1999. Document ad/99-02-05. `http://www.omg.org`.

[16] Parasoft. jContract. Web site, Parasoft, 1996.

[17] R. Kramer. iContract - The Java Design by Contract Tool. Web site, iContract, 1999.

[18] Sun Microsystems. *Enterprise Java Beans*. `http://www.javasoft.com/products/ejb`.

[19] JBoss team. JBoss Aspect Oriented Programming. *http://www.jboss.org/*, 2002. developers/projects/jboss/aop.

[20] E. Truyen, B. Vanhaute, W. Joosen, P. Verbaeten, and B. Joergensen. Dynamic and selective combination of extensions in component-based applications. In *Proceedings of ICSE'01*, 2001.

[21] A. Vasseur. Dynamic aop and runtime weaving for java - how does aspectwerkz address it ? In *AOSD 2004, Dynamic AOP WorkShop (March 2004)*, March 2004.